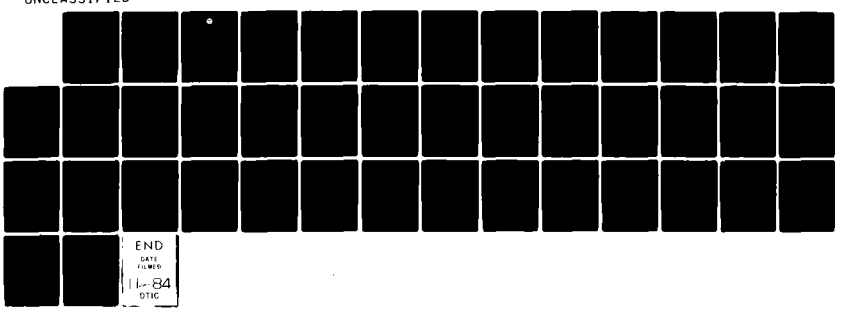
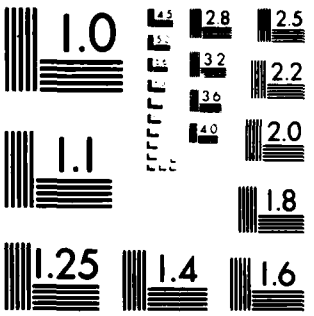
AD-A146 417 ALGEBRAIC TECHNIQUES IN SYSTOLIC ARRAY DESIGN(U) NAVAL 1/1
OCEAN SYSTEMS CENTER SAN DIEGO CA G E CARLSSON ET AL.
FEB 84 NOSC-TR-942

UNCLASSIFIED F/G 12/1 NL



END
DATE FILMED
11-84
DTIC



(12)

NOSC TR 942

NOSC TR 942

Technical Report 942

ALGEBRAIC TECHNIQUES IN SYSTOLIC ARRAY DESIGN

G. E. Carlsson
Department of Mathematics, UCSD
H. B. Sexton, M. J. Shensa
NOSC, Code 6322
C. G. Wright
Department of Mathematics, Duke University

February 1984
Final Report

NOSC

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152

NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN
Commander

R.M. HILLYER
Technical Director

Administrative Information

The research described in this report was done during fiscal year 1983 under the Independent Research/Independent Exploratory Development Program. The authors worked collectively, but took responsibility for separate chapters. The work was funded under program element 61152N, project ZR00001, task area ZR0000101, and task area 632Z96.

Released by	Under authority of
P. M. Reeves, Head	R. H. Hearn, Head
Electronics Division	Fleet Engineering Department

Acknowledgements

We wish to thank Debbie Watson of NOSC COde 632 for her expert typing of this report and her patience and ingenuity in making revisions. We wish to thank Jim Zaun, Code 632, for his help in subduing **troff** and **mm**, and in getting this report into a presentable form. Finally, we wish to thank Dr. Eugene Cooper, Code 013, for his continued support of this work: financially, as head of NOSC's IR/IED program, and morally, by his encouragement and interest in our project.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR 942	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Ocean Systems Center 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) San Diego, CA 92152	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Independent Research/ Independent Exploratory Development 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 61152N; PROJECT NO: ZR00001; TASK NO: ZR0000101; WORK UNIT NO: 632Z96

11. TITLE (Include Security Classification)
ALGEBRAIC TECHNIQUES IN SYSTOLIC ARRAY DESIGN

12. PERSONAL AUTHOR(S)
G.E. Carlsson, UCSD; H.B. Sexton and M.J. Shensa, NOSC; C.G. Wright, Duke U.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Oct 82 TO Sep 83	February 1984	39

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel Processing; Graphy Theory; Communications Network; Automata Theory; Hardware Description Language

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes the work of the FY83 NOSC IR project "Algebraic Techniques in Systolic Array Design." The goal of the project was to develop an abstract mathematical framework general enough to include the standard mesh-connected architectures as well as more complex ones such as the cube-connected cycles, but restricted enough that the algebraic techniques used (sporadically) in the analysis of systolic arrays could be generalized and applied.

It is our contention that node-transitive networks, based especially on Cayley graphs, offer such a framework, though whether such networks provide a truly viable theory for modelling synchronous parallel computations remains to be seen. This report describes how to restrict the theory of synchronous parallel architectures to such networks, as well as demonstrating the applicability of algebraic techniques to such networks. It also describes various tools, such as the programming language MHDL, which were developed to aid in the study of these networks.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified	Agency Accession
22a. NAME OF RESPONSIBLE INDIVIDUAL H.B. Sexton	22b. TELEPHONE (Include Area Code) (619) 225-2287	22c. OFFICE SYMBOL

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Contents

1. Introduction

1.1 Background

Originally, when we began this project we intended to develop a decomposition theory for implementing parallel algorithms on "standard" mesh-connected systolic arrays. We believed we had good reasons for feeling that this problem was a highly algebraic one, and that various ideas could be used to arrive at systematic design procedures. We still agree with this point of view, but the emphasis of our work has changed considerably, for reasons which I hope to make clear.

There were two problems which we hoped to solve by our algebraic approach. The first is the problem of sorting N elements by pairwise exchanges, and the second, that of tridiagonalizing a symmetric matrix by Givens rotations. (These problems are discussed further in Chapters 2 and 3, respectively.) While we were able to find an optimal solution to the first problem for a linear array, we were unable to do so for a rectangular one, and we failed similarly for the tridiagonalization problem. From these experiences, it became clear to us that there were extremely difficult "constrained parallel complexity problems" which needed to be solved. That is, while there is a reasonably well-developed theory of computational complexity for single processor computers and some theory for unrestricted parallel computation, there is almost no theory for the complex computations subject to communication constraints imposed by a processor network.

It is our hope that the concepts which we are now exploring will help to understand problems of parallel computation, especially ones which arise in parallel architectures with limited inter-processor communication. We would, of course, hope to develop provably optimal architectures with such a theory, but this seems to be extremely difficult for all but the simplest of problems, so for now our work focuses on developing "better" architectures. First, we are searching for a reasonable framework in which to formulate a version of such a constrained theory of complexity and, in particular, for a class of communication networks general enough to contain optimal or near optimal networks for standard problems such as sorting, matrix multiplication, FFT computation, and for which the fabrication costs are acceptable. It is this which has led us to propose Cayley networks (see Chapter 2) as a possible class, but we feel that it is far too soon to have any confidence that this is a final choice. Second, we are attempting to formulate mathematical questions which are equivalent to determining lower bounds on the time complexity of the sorting and tridiagonalization problems for a given class of arrays, and to solve these problems for mesh-connected arrays. Third, we

are attempting to use the insights gained in our theoretical work to develop architectures to implement various algorithms in efficient ways. Finally, in the course of all this work we are attempting to develop tools to aid ourselves and others in future work in these areas; our major such tool, so far, is our high-level simulation language MHDL, which is still in its earliest stages of development.

1.2 Chapter Summaries

Preliminary Concepts: Chapter 2 gives the definitions and notations to be used throughout the report. In an attempt to be at least formally self-contained, we shall give definitions of a number of elementary mathematical structures, such as groups and graphs, but we shall provide relatively little intuition for many of the concepts we define. We will attempt to give adequate references for all concepts which we use, however.

Matrix Tridiagonalization: Chapter 3 concerns the problem of tridiagonalizing a real symmetric matrix by using Given's rotations acting in parallel. We derive a lower bound for a class of TD algorithms.

Regular Graphs and Cayley Graphs: Chapter 4 contains some of the results we obtained in our study of graphs. In particular, we describe the results of our heuristic search methods for regular graphs, and introduce the concept of a Cayley graph.

MHDL: Finally, in Chapter 5 we describe the Modular Hardware Description Language. This is a high-level simulation language which we have defined for testing algorithms for our general class of Synchronized Modular Networks. While this language is currently working "as advertised," it is very much under development and subject to drastic change without notice.

1.3 Remarks

Some features of the organization of this report seem worth comment. Rather than have an index of notation, as is more typical of mathematical work, we have included a list of symbols under the heading "Notation" in the regular index. Also, instead of a single bibliography, each chapter has a closing section containing references.[1] Consequently, any citation within a chapter is to one of the references listed at the end of the chapter. Finally, it may seem to the reader that it is rather absurd to call the major divisions of such a short document "Chapters"; rest assured that it does to us, as well, but for reasons too boring to mention here it was expedient.

[1] This is due, principally, to having the various chapters written by the different authors.

2. Preliminary Concepts

2.1 Introduction

This chapter introduces the principal mathematical concepts of this report. As this document is intended to serve as a basic reference for our later work, concepts will occasionally be introduced somewhat baldly, and in some cases receive no further development here.

At the most abstract level, we are interested in developing a theory for a certain class of automata, which we call *synchronized modular networks*, or SMNs. These automata are defined in Section 2.2, but aside from some trivial results which we prove there, no further mention of these objects will be made here. However, being mathematicians, we need to know precisely what we are talking about, even when it is irrelevant, so we present these definitions.

Section 2.3 defines various types of combinatorial graphs which were studied as possible communication networks and parallel computation architectures. None of these ideas is new with us and any originality in the presentation is the result of the ubiquitous random processes which influence all our lives.

Section 2.4 discusses briefly the concept of communications compatibility between architectures and algorithms, and illustrates this concept with some examples drawn from parallel sorting problems.

2.2 Synchronized Modular Networks

The concepts discussed here regarding Synchronized Modular Networks are drawn from the theory of automata. They are given here simply to provide a precise basis for the rest of our mathematical work, rather than as a starting point for our investigations, as we have been unable to prove much of interest within this very general framework.

Definition 2.2.1: An *abstract* machine M is a quintuple (I,X,O,δ,β) where

$$I = I_1 \times \cdots \times I_{k_M} ,$$

$$O = O_1 \times \cdots \times O_{\ell_M} ,$$

$$\delta: I \times X \rightarrow X,$$

and

$$\beta: X \rightarrow O.$$

I, O, and X are referred to as the input, output, and state sets, respectively, of M. δ is the next state or state transition function, and β is the output function. Intuitively, the machine cycles as follows:

M is in some state x_1, receives input a_1, moves to internal state $\delta(a_1,x_1) = x_2$, outputs $\beta(x_2)$ and waits in state x_2 for more input.

We say state y is reachable from state x if there exist $a_1, \ldots, a_n \epsilon I$, $x_1, \ldots, x_n \epsilon X$ with $x = x_1$, $x_{i+1} = \delta(a_i,x_i)$, for $i = 1, \ldots, n-1$ and $y = \delta(a_n,x_n)$. Assume all states are reachable from one another, and we shall suppose that X has a designated x_s which we shall call the initial state.

Also, we associate with a machine M the sets $l_I = \{ i_1, \ldots, i_{k_M} \}$ and $l_O = \{ o_1, \ldots, o_{k'_M} \}$, which we call the set of input and output leads, respectively. If we have some indexed family M_α of machines, the leads will be denoted $l_I(\alpha), \ldots, o_j(\alpha)$, or $l_{I_\alpha}, l_{I,\alpha}, o_{j,\alpha}$, etc.

For our discussion we assume the existence of some set P of primitive machines, or processing elements (PEs).

Definition 2.2.2: Let M be some set of abstract machines. Then $N = (I,F,I^E,O^E,c)$ is a **synchronized M-modular machine**, or an **SMN** of **M-type**, iff the following hold:

a. I is a finite index set.

b. F: $I \rightarrow M$.

c. I^E and O^E are input and output sets, respectively.[2]

d. c is a bijection where

$$c : l_I(E) \bigcup \left\{ \bigcup_{\alpha \epsilon I} l_O(\alpha) \right\} \rightarrow l_O(E) \bigcup \left\{ \bigcup_{\alpha \epsilon I} l_I(\alpha) \right\},$$

such that the intersection of $c(l_I(E))$ with $l_O(E)$ is empty.

We denote by c_α the projection of c into $l_I(\alpha)$, and c_O the projection of c into $l_O(E)$.

c has a natural extension from leads to

$$c : I^E \times \left\{ \underset{\alpha}{\times} O_\alpha \right\} \rightarrow O^E \times \left\{ \underset{\alpha}{\times} I_\alpha \right\}.$$

We use this map without further comment.

[2] That is, as we said before, $I^E = I_1^E \times \cdots \times I_{k_N}^E$ etc. The leads of I^E and O^E are denoted $l_I(E)$, $O_I(e)$, etc. We shall assume that E,I are not in I

Proposition 2.2.1: If N is an SMN of M-type, then N is an abstract machine.

Proof: We associate N with the machine

$$(I^E, \underset{\alpha}{\times} X_\alpha, O^E, \delta_N, \beta_N),$$

where[3] $\beta_M(x) = c_O(<\beta_\alpha(x_\alpha)>)$. Next, if $a^E \epsilon I^E$, and $<x>$ belong to $X = \underset{\alpha}{\times} X_\alpha$, and if δ_α is the state transition function for M_α, then

$$\delta_M(a^E, <x>)_\alpha = \delta_\alpha(c_\alpha(a^E, <\beta_\alpha(x_\alpha)>)).$$

Definition 2.2.3: If P is a set of machines, then P' denotes the set of SMNs of P-type.

Proposition 2.2.2: $P'' = P'$.

Proof: It follows directly from the definition and the previous proposition.

Remarks:

a. We generally assume that we inhabit the universe of SMNs of P-type for some fixed P, and suppress all mention of P.

b. We refer to a *conjunction* of two SMNs, M_1 and M_2, as an SMN formed by connecting some outputs of M_2, and vice versa. A *cascade* of M_1 with M_2 is a conjunction where no outputs of M_2 are connected to M_1.

c. It is obvious that every SMN with n PEs can be formed by the conjunction of an $n-1$ element SMN with a single PE.

2.3 Graphs and Networks

Defining a number of standard concepts from the theory of combinatorial graphs, we use them to relate to problems in networks in this and the next section. As these concepts are very abstract, the mathematical questions about networks which can be formulated in terms of them are much simpler than the actual details which would arise in practical implementation of a network of synchronized processors. However, these questions are already extremely difficult in many cases, and in some sense we feel that their generality helps to ensure their usefulness. Thus, while intelligent readers will doubtless see many ways in which these formulations are inadequate in helping to understand real parallel processing networks, we hope that they will feel the solution of problems posed in this report will be of real use in gaining such understanding.

We begin by defining a graph (also known as an undirected graph). A *graph* Γ is a pair (V,E), where V is a set of points known as *vertices* or *nodes*, and E, the *edge* set, is a set of (unordered) pairs of of nodes $\{v,w\}$, where v,w belong to V. We think of the edge

[3] Notice that we make use of the fact that the preimage of c_O is a subset of $\bigcup_\alpha I_O(\alpha)$.

$\{v,w\}$ as being a line segment connecting the points v and w, and we say that this edge is *incident to* v (and w). This edge[4] will usually be denoted vw.

The *degree* of a node, or vertex, v in Γ is the number of edges in Γ incident to v. If all nodes in Γ have the same degree, we say that Γ is *regular*.

A *path* of *length* n from x to y is a sequence of vertices $\phi=\{v_0, \ldots, v_n\}$ in Γ, so $v_0=x$, $v_n=y$, and so for each i, v_iv_{i+1} is an edge of Γ. Clearly, this is the same as a sequence of wires connecting the processors corresponding to x and y. (It might be that v_i and v_{i+2} are the same, corresponding to a path which doubles back. We do not consider the case where v_i and v_{i+1} are the same; that is, we don't admit the case of an edge connecting a vertex to itself.)

The *distance* between distinct nodes v and w is the minimum length of paths between them. (The distance from v to itself is taken to be zero.) We denote this distance by $d(x,y)$. By the *diameter* of the array, we mean

$$\Delta(\Gamma) = \max_{x,y\in\Gamma} d(x,y).$$

where the max is taken over all pairs of vertices in Γ.

We now want to define a restricted type of graph which is of interest partly because it possesses a high degree of regularity. First, however, we give the definition of a type of algebraic construction known as a group.

Let G be a set of points, or *elements*, and let $\times$ be an operation which takes ordered pairs of elements (a,b) to a third element, denoted $a\times b$ or simply ab. We say that there is a distinguished element known as the *identity element*, denoted by e, provided $e\times a = a\times e = a$, for all a in G. We say that a has an *inverse*, denoted by a^{-1}, provided $a^{-1}\times a = a\times a^{-1} = e$. We say that $\times$ is *associative* provided $a\times(b\times c) = (a\times b)\times c$, for all a, b, and c in G.

A *group* is a set G and an operation $\times$ such that $\times$ is associative and every element of G has an inverse.

Some of the simplest examples of groups are the *cyclic*[5] *groups*, denoted Z/n, which are the integers $\{0, \ldots, n-1\}$ with the operation being addition modulo n.

Definition 2.3.1: A *Cayley graph* is a graph constructed as follows: Let the point set of the graph G be some finite group, also denoted by G. Let Ω be some set of elements of G where any element in G can be written as the product of elements of Ω (that is, Ω generates G), and $\Omega^{-1} = \Omega$. Then every element g in G is connected to all elements of the form ωg, where ω is in Ω, and only those elements. We denote the Cayley graph associated with the pair (G,Ω) by $\Gamma(G,\Omega)$. It is easy to see that such a graph is of a type known as vertex transitive (defined below), and it is obvious that the degree is the number of

[4] It seems worth remarking here that one could make all these definitions with *ordered* pairs of vertices, giving rise to what is known as a *directed* graph. In networks where the flow of information is non-symmetric, such a formulation would seem called for

[5] The operation in these groups is known as "clock arithmetic" to those upon whom "the NEW math" was inflicted.

elements in Ω. (Biggs [3] provides extensive information about such graphs.)

This class of graphs is very large (it is of course infinite, but it is also large in a more meaningful sense) in that it contains a number of examples of interesting architectures, and many more can be obtained by simple constructions using these arrays. As one example, we show in Chapter 4 how to construct a class of networks known as the Cube-Connected Cycles in the manner described above.

Definition 2.3.2: The concept of the symmetry of a graph is precise. Let Γ be a graph with nodes labeled {1,...,n}. Then, a permutation τ of the integers {1,...,n} is called an *automorphism* of Γ if and only if, for all i and j, $\tau(i)$ is connected, or adjacent, to $\tau(j)$ if, and only if, i is connected to j. Thus, as far as Γ is concerned, i and $\tau(i)$ "look" exactly the same if τ is an automorphism. We say that Γ is *node*, or *vertex*, *transitive*, provided for all nodes i and j there is an automorphism τ sending i to j.

2.4 Sorting

We now relate the idea of data dependence for an algorithm to communication constraints of a network by considering methods for sorting which conform to the communication restrictions imposed by our graph. In general, a sort may be thought of as a permutation ρ, where the contents of the i-th node are sent, ultimately, to the $\rho(i)$-th node. (Here we rather naturally suppose the it was desired to get the contents of the i-th node to the $\rho(i)$-th node. If we imagine the initial contents of the i-th node to be $\rho(i)$, then our sorting algorithm is in effect computing the inverse of ρ.) Of course the particular permutation "chosen by the algorithm" will depend on the initial contents of various nodes, i.e., the initial state of the array. We shall say that a sorting algorithm is consistent with a graph Γ provided it generates a sequence of permutations ρ_k, where for each j,

$$\rho(j) = (\rho_k \cdots \rho_1)(j), \tag{2.1}$$

and for each k and j, $\rho_k(j)$ is connected to j. (That is, the permutation ρ is performed in k steps, and at the i-th step the contents of the j-th node are sent to some node $\rho_i(j)$ which is connected to j.) We shall consider here only consistent algorithms where each of the ρ_i may be taken to be a transposition or the identity.

We say that the *time complexity* of a sequence of permutations, as in Equation 2.1, is the minimum number of terms into which ρ can be decomposed, where the terms are of the form $\rho_i \cdots \rho_{i+l}$ and the various ρ_j in a given term commute. (We call such a term a *time-step*. Since permutations commute if, and only if, they act on disjoint sets, a time-step is some collection of permutations which can be performed concurrently.) Finally, the *time complexity* of an algorithm is the maximum time complexity of the permutations ρ generated by the algorithm, where this maximum is taken over all initial states of the array.

One fact which is completely obvious is that, by this definition, the time complexity of any consistent algorithm is at least as great as the diameter of the underlying graph, since each time-step moves the value at the i-th node over at most one edge in the graph. Consequently, the sorting problem for a systolic array[6] with N elements is of time complexity at least $O(N)$. However, the best[7] results of which we are aware sort in $O(N)$ and in particular this can be realized for a wide class of arrays.

However, it is not enough to make the diameter small, as consideration of a 2-tree quickly shows. The difficulty here is that of congestion, i.e., many shortest paths pass through the same node; every path from the left half of the tree to the right half of the tree must pass through the root, and so the complexity must be at least $O(N)$ for an N element tree.

There are, of course, many other considerations in assessing the costs of various algorithms/arrays for sorting. It seems very desirable for the algorithm, and hence the array, to have a sufficiently simple regular structure so a theoretical verification of the algorithm would be feasible. Also, it seems discordant to the spirit of distributed processing to have the permutations ρ_k to have general dependence on the current state, especially when they are as simple as transpositions. In this last case, it seems most natural to require that ρ_k depends only on the states of some pair of nodes i_k and j_k, and that the particular nodes depend only on the stage k. With these restrictions, the best sorting algorithms known require[8] $O(\log^2(N))$. Connections between sorting and graph theoretic problems will be discussed further in Chapter 4.

2.5 **Matrix Tridiagonalization**

This section contains definitions for concepts needed in Chapter 3. More detailed information on these topics may be found in the excellent book by Parlett, Reference [4].

Definition 2.5.1: Let $A = (a_{ij})$ be an $n \times n$ real symmetric matrix. We say that A is *tridiagonal* if, and only if, $a_{ij} = 0$ for $|i-j| > 1$. Let $R_2(\theta)$ denote the plane rotation

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Then, notice that $R_2(\theta)^{-1} = R_2(-\theta) = R_2(\theta)^t = R_2(\theta)^*$. In n-space let $R_n(i,j,\theta)$ denote the rotation which is the identity on the orthogonal complement of the plane given by the i, j basis vectors, and is $R_2(\theta)$ in this plane. Finally, for $i,k \neq j$, let $G(i,j,k)$ denote a transformation

$$A \rightarrow R(i,j,\theta)AR(i,j,-\theta) = A',$$

[6] See reference [1] for a definition of *systolic array*.

[7] See Knuth [2], Section 5.3.4.

[8] Again, see [2].

such that $A'_{jk} = 0$. It is easy to see that $G(i,j,k)$ is unique up to the sign of θ if i, j, k are distinct, and if $k = i$ then $G(i,j,k)$ is given by θ in $\{ \phi, -\phi, \frac{\pi}{2}-\phi, \phi-\frac{\pi}{2} \}$, for a unique ϕ. This transformation is referred to as a *Givens rotation.* For A Hermitian, $G(i,j,k)$ is defined in a completely analogous fashion, and there are similar types of simple ambiguities in its definition exist, although θ may now be complex. See Reference [4], especially Chapter 6, Section 4, for a more detailed exposition.

2.6 References

[1]. Mead, C., and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[2]. Knuth, D. E., *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Addison-Wesley, 1973.

[3]. Biggs, N. L., *Algebraic Graph Theory*, Cambridge University Press, 1974.

[4]. Parlett, B., *The Symmetric Eigenvalue Problem*, Prentice-Hall, 1980.

3. Matrix Tridiagonalization

3.1 Introduction

The increasing availability of VLSI (Very Large Scale Integration) devices and specialized computer architectures has led to a surge of interest in algorithms which utilize parallel processing. A large number of these algorithms has been directed towards solving classical problems in linear algebra with notable successes in solving linear equations and performing matrix operations (References [1]-[3]). However, the computation of the eigenvalues and eigenvectors of symmetric (Hermitian) matrices remains an area in which results have been somewhat less effective than one might hope. The standard approach, which also lends itself to parallel architectures, is tridiagonalization followed by the QR algorithm ([4]-[7]). The major stumbling block in a fast implementation of such procedures is the tridiagonalization, which, under current methods, requires a time of $O(N^2)$ to reduce an $N \times N$ matrix, in sharp contrast to $O(\log N)^2$ for each iteration of the QR algorithm ([5]-[7]).

Thus, it is of interest to establish bounds on the ability of parallel processing to speed the reduction of a matrix to tridiagonal form. Unfortunately, the analysis given here is of a very restricted nature, as it assumes that the algorithm doesn't make non-zero any entries which have been previously set to zero. However, while this requirement may seem absurd, many of the proposed methods which were circulating informally at the time this work was being done satisfied this condition. Therefore, while this analysis offers little for the general case, it at least explains why those proposed methods had such poor performance, and why the current systolic methods which work in $O(N^{3/2})$ time are of such different character. (See Reference [8].)

3.2 Problem Statement

Let A denote an N-dimensional symmetric (Hermitian) matrix. A Givens rotation $G(i_1,i_2,j)$ is the essentially unique rotation in the plane of the i_1 and i_2 coordinate axes which results in $A'_{i_2 j} = 0$ ([4]). Note that such a transformation is represented by an orthogonal (unitary) matrix R where $A \rightarrow A' = RAR^{-1}$. The effects of a Givens rotation in creating or destroying matrix zeros may be characterized as follows (see Figure 3.1):

a. Only rows i_1 and i_2 and columns i_1 and i_2 are effected.

---

[2] $O(N)$ for systolic arrays

b. A zero is produced at element $A_{i_2 j}$ and symmetrically at $A_{j i_2}$.

c. For any $k \neq i_1, i_2$, if $A_{i_1 k} = A_{i_2 k} = 0$ prior to rotation, then they remain zero after rotation. (By symmetry this also holds for $A_{k i_1} = 0$).

Thus, with the exception of cases b and c, all elements of rows and columns i_1 and i_2 in the resulting matrix will be generically non-zero.

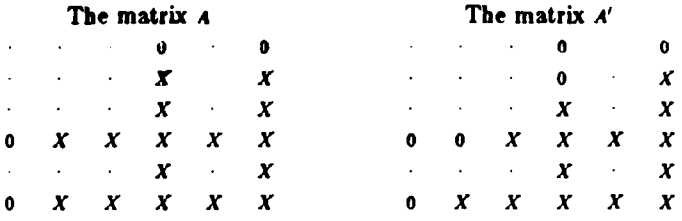


Figure 3.1 The Givens rotation $G(6,4,2)$. Elements represented by dots are unaffected; those represented by Xs are changed, and the 0s behave as shown.

Next consider the problem of reducing a symmetric matrix A to tridiagonal form using Givens rotations. We also wish to make use of parallel processing, so we allow several rotations to be performed simultaneously, provided they involve disjoint sets[10] of rows. In that case, the corresponding matrices R (equivalently the angles of rotation) will be independent of each other, and no element of A will be affected by more than two Givens rotations. It then follows that with sufficiently many processors such a set of rotations may be performed in $O(1)$ time. Our principal result is the following theorem:

Theorem 1: Suppose we are given an algorithm for tridiagonalizing an $N \times N$ symmetric (Hermitian) matrix by, possibly concurrent, Givens rotations as described above. If the algorithm never replaces a generic zero by a generic non-zero, then it requires at least $O(N \log N)$ time steps. Furthermore, this lower bound is realizable.

The proof of Theorem 1 will be divided between the next two sections.

3.3 An O(N log N) Algorithm

In the interest of simplicity, in Sections 3.3 and 3.4, our discussion is restricted to a description of the elements below the diagonal. Since the matrix is symmetric (Hermitian), there is essentially no loss of generality. The stipulated algorithm proceeds by zeroing one column at a time starting with $j = 1$ and following with $j = 2$, etc. It is clear from property (c) that if **all** sub-tridiagonal elements A_{ik} are zero for $k < j$, they will remain zero under Givens rotations of the form $G(i_1, i_2, j)$ where $i_1, i_2 \geq j+1$.

[10] More formally, a collection of rotations $(G(i_1,i_2,j), G(i_3,i_4,k),\ldots$ is permitted to be performed in parallel provided the sets $\{i_1,i_2\}, \{i_3,i_4\},\ldots$ are disjoint.

We next show that column j may be zeroed in $\log_2(N-j-1)$ steps. By pairing rows $j+1$ through N we may simultaneously zero half the sub-tridiagonal elements of column j (rows $j+2$ to N). We then repeat the process for the remaining half of the rows, which still have non-zero elements in column j. Continuing in this fashion, we find that all elements of column j (i.e., A_{ij} for $i \geq j+2$) have been reduced in q steps where

$$N-j-2 \leq (N-j-1)\left(\frac{1}{2}+\frac{1}{4}+\cdots\frac{1}{2^q}\right)$$

or

$$1-\frac{1}{N-j-1} \leq 1-\frac{1}{2^q}.$$

Thus, it is sufficient that

$$q \geq \log_2(N-j-1). \tag{3.1}$$

Finally, summing over j and noting that q must be an integer we find that our algorithm takes

$$\sum_{j=1}^{N-2}(\log_2(N-j-1)+1) = \log_2(N-2)! + N - 1 \tag{3.2}$$

steps which, by Stirling's formula, is $O(N\log_2 N)$. This proves existence.

We remark that such an algorithm requires communication across the entire matrix. If we restrict ourselves to "local" connections as in systolic arrays, it is not generally possible to achieve the same speed. Pipelining still enables us to perform Givens rotations in $O(1)$ time (References [3] and [6]). However, it is not difficult to see that if only adjacent matrix elements may communicate (i.e., only rotations of the form $G(i_1,i_1\pm1,j)$ are allowed), there is essentially only one algorithm for reducing a column. It must start at the bottom, $G(N-1,N,j)$, and proceed up, ending at $G(j+1,j+2,j)$. We then find using Lemma 1 of Section 3.4, that for algorithms with "local communication" the best that can be done is $O(N^2)$ time steps. (This statement should not be taken too rigorously since we have not really defined the term "local".)

a.

$$\begin{matrix} X & & & & & \\ X & X & & & & \\ 0 & X & X & & & \\ 0 & 0 & X & X & & \\ 0 & 0 & 0 & X & X & \\ 0 & 0 & 0 & 0 & X & X \end{matrix}$$

The final state of matrix A.

b.

$$\begin{matrix} X & & & & & \\ X & X & & & & \\ 0 & X & X & & & \\ X & 0 & X & X & & \\ 0 & 0 & 0 & X & X & \\ 0 & 0 & 0 & 0 & X & X \end{matrix}$$

A hypothesized previous state.

Figure 3.2

3.4 Derivation of a Lower Bound

To complete the proof of Theorem 1, we establish the following lemma:

Lemma 1: An algorithm of the above type (tridiagonalization by Givens rotations) must proceed by the successive annihilation of columns. More precisely, if the algorithm is to take a minimal number of time steps, and not make generic zeros non-zero, column j must be placed in tridiagonal form prior to column $j+1$.

Of course, one can zero elements in column $l > j$ prior to j, but this lemma states that one will eventually have to make non-zero some element of column l.

We first note that, given any algorithm which employs parallel Givens rotations, we may assume the existence of an equivalent algorithm with Givens rotations in sequence. (Simply order the concurrent rotations performed at each step in an arbitrary manner.) The proof, then, is obtained by induction, starting from the tridiagonalized matrix and working backwards. To motivate this method let us consider the tridiagonal matrix pictured in Figure 3.2a. The final zero placed by the algorithm could only be element $A_{5,3}$. The creation of any other zero would also have created a non-zero element. For example, zeroing element $A_{4,1}$ of Figure 3.2b by a Givens rotation with row 5, $G(5,4,1)$, creates a non-zero element at $A_{5,1}$. Alternatively, the use of row 3 destroys (by its action on column 3) $A_{5,3}$. Similarly, the use of row 2 destroys $A_{4,2}$ and $A_{5,2}$, and row 1 would destroy $A_{3,1}$ and $A_{5,1}$. In other words, we conclude that the last stage of the algorithm must have consisted of the single Givens rotation $G(4,5,3)$ using row 4 to zero the element $A_{5,4}$. We now proceed to the general proof.

Pf. of Lemma 1: Suppose that at some stage s of the algorithm we have the following situation below the diagonal (where $j \leq N-2$): columns 1 to j have their final tridiagonal structure; column $j+1$ has at least one "generic" zero; and columns greater than $j+1$ are arbitrary. (This situation is pictured in Figure 3.3 for $j = 2$ and $N = 8$.) Then the previous stage $s-1$ could not have had a non-zero off-tridiagonal element in column j, say $A_{l,j}$ with $l \geq j+2$, because zeroing that element by a Givens rotation would have involved *either*

a. The interaction of row l with another row $r \geq j+2$ which would create a non-zero entry at $A_{r,j}$ destroying the tridiagonal structure for columns 1 to j;

or

b. The interaction of row l with a different row $r < j+2$, which implies the interaction of column l with column r. This interaction would create a non-zero entry at $A_{l,r}$ and either destroy the tridiagonal structure (if $r < j+1$) or remove a zero from column $j+1$ (if $r = j+1$).

These aspects are illustrated in Figure 3.3 for $l = 6$. It now follows by induction on j for $j = n-1, n-2, \ldots, 1$ that column $j-1$ must have been completely reduced prior to initiating the final reduction of column j. Thus, an algorithm of the type specified must proceed column by column; the reduction of any matrix elements

outside such a sequence results in creation of new generic non-zeros.

To complete the proof of Theorem 1, we note that in reducing column j we may only use rows $j+1$ through N (otherwise we introduce non-zeros in the column through mechanism (b)). Similarly, we may not use a row r with a zero in the j^{th} column to reduce some other row since the zero A_{rj} will be destroyed. Finally, the condition that concurrent Givens rotations be performed on disjoint pairs of rows implies that at most one-half the non-zero entries may be reduced in one stage of the algorithm. This restricts us to the situation of Equation (3.1); i.e., $O(\log(N-j-1))$ steps are necessary to reduce column j. Equation (3.2) then implies that the entire tridiagonalization takes at least $O(N \log N)$ time steps.

a.

$$\begin{matrix}
X & & & & & & & \\
X & X & & & & & & \\
0 & X & X & & & & & \\
0 & 0 & X & X & & & & \\
0 & 0 & 0 & X & X & & & \\
0 & 0 & 0 & X & X & X & & \\
0 & 0 & X & X & X & X & X & \\
0 & 0 & 0 & X & X & X & X & X
\end{matrix}$$

A possible state of the matrix at stage s of the algorithm. This corresponds to the case $j=2$ of the text.

b.

$$\begin{matrix}
X & & & & & & & \\
X & X & & & & & & \\
0 & X & X & & & & & \\
0 & 0 & X & X & & & & \\
0 & 0 & 0 & X & X & & & \\
0 & X & 0 & X & X & X & & \\
0 & 0 & X & X & X & X & X & \\
0 & 0 & 0 & X & X & X & X & X
\end{matrix}$$

A hypothetical situation one Givens rotation short of stage s, assuming stage s created a zero at $A_{6,2}$. This is impossible as: **(I)** the interaction of row 6 with any row $\geq j+2$ creates a new X in column 2, and **(II)** the interaction of row 6 with row $r<j+2$ implies interaction with column 6 with column r which yields a new X in column r.

Figure 3.3

3.5 Remarks

By restricting our arguments to elements below the diagonal, we have implicitly assumed that $i_2 > j$ in $G[i_1,i_2,j]$. If this restriction is relaxed, we find that the last Givens rotation may have zeroed either $A_{3,1}$ (result by symmetry of applying $G[\cdot,1,3]$) or $A_{N,N-2}$. As a consequence, the reduction may actually proceed by simultaneously reducing rows (bottom to top) and columns (left to right). The number of time steps still remains $O(N \log N)$, however.

The astute reader may also have noted that our arguments assume the elements of the two subdiagonals are generically non-zero and justifiably wonder whether some appropriate zeroing of these elements, or others, at intermediate stages could have a positive effect on the algorithm. That this is so has been shown in Reference [8], by R. Schreiber.

3.6 References

[1]. Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison Wesley, Reading MA, 1980; especially Chapter 8.

[2]. Kung, H. T. and Leiserson, Charles E., Systolic Arrays (for VLSI), *Sparse Matrix Proceedings*, SIAM, pp. 256-282, 1978.

[3]. Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, 20, 4 Oct 1978, pp. 740-777.

[4]. Parlett, B., *The Symmetric Eigenvalue Problem*, Prentice-Hall, 1980.

[5]. Sameh, A., and Kuck, D., "A Parallel QR Algorithm for Symmetric Tridiagonal Matrices," IEEE Trans. on Computers, C-26, 2 Feb 1977, pp. 147-153.

[6]. Schreiber, R., "Systolic Arrays for Eigenvalue Computation," SPIE Technical Symposium East, SPIE Vol 341, May 1982.

[7]. Heller, D., and Ipsen, I., "Systolic Networks for Orthogonal Decompositions with Applications," Computer Science Dept., CS-81-18, Penn State Univ., Aug 1981.

[8]. Schreiber, R., "Computing Generalized Inverses and Eigenvalues of Symmetric Matrices using Systolic Arrays", Comp. Sc. Tech. Rep., Numerical Analysis Project, #NA-83-03, Nov 1983, Stanford Univ., Stanford CA.

4. Regular and Cayley Graphs

4.1 Introduction

It is clear that communication time between processors is one of the most severe limiting factors in designing high speed parallel computers. Under these circumstances, it is obviously important to be able to design networks of processors in which communication time is as short as possible. A mathematical version of this design problem is the problem of constructing graphs of given fixed degree and number of vertices with small diameter. (See Chapter 2 for definitions of these terms.) We have attacked this problem from two directions. First, we have constructed a heuristic algorithm which finds graphs with small diameter, and implemented it on a computer. The program is written in the C-language. We have compared the results with previous best known results. Secondly, we have studied a collection of graphs which have compact and systematic descriptions, the so-called Cayley graphs. Routing algorithms for these graphs is easy to specify. We have given a crude comparison of their diameter with that of a theoretical bound, and studied a specific class of them, the "modified cube-connected cycles."

4.2 Summary of Results

The results obtained may be summarized as follows.

a. The heuristic algorithm we constructed is an improvement over all previous algorithms of its kind. In particular, we improved many of the best known values for dense graphs of given degree and diameter. A complete description is given in Section 4.2.2, where Figure 4.1 shows all the improved values we obtained.

b. Our algorithm does not find the densest known graphs in cases where they are constructed using systematic combinatorial constructions. This suggests that one should study a restricted class of graphs that has systematic descriptions.

c. The symmetric groups S_n admit graph structures whose diameters approach the theoretical bound (Moore bound) arbitrarily well as $n \to \infty$. This suggests that they should be studied much more carefully as a possible source of efficient communications networks.

d. A slight modification of the cube-connected cycles of Reference [1] produces an infinite family of graphs whose diameter grows as $2\log_2(K)$, where K is the number of points in the graph. This compares favorably with $5/2\log_2(K)$, which is the diameter of the cube-connected cycles.

e. A layout is given for these modified cube-connected cycles, whose area is 3/2 times the area of the cube-connected cycles with the same number of nodes. The VLSI measure of complexity, AT^2, is thus slightly decreased by a factor of 24/25.

These results are substantiated by running the heuristic program which we have designed, and by theoretical analysis contained in Section 4.2.

4.3 Detailed Description of Results

4.3.1 Measures of Communication Time.

Throughout, we are interested in arrays of "processors" connected by "wires." The nature of these processors is not specified, because we want to study the general problem of communication time, without restricting to a specific situation. We formalize this notion by considering graphs Γ, where the vertices of the graphs correspond to processors and edges to wires. We will suppose that the array functions in such a way at every time information is allowed to flow along one wire. The time required to move along one wire is presumed to be constant. By the *distance* between two processors, or the corresponding vertices x and y, we mean the length of the shortest path in Γ from x to y. (These and other standard terms from graph theory are defined in Chapter 2.) From this point on, we no longer speak of the arrays of processors but only of their corresponding graphs. We wish to study the problem of designing graphs with a given number of vertices, having small diameter. Of course, with no constraints on the graph, this is a trivial problem since complete graphs all have diameter 1. However, technology dictates that the number of edges from each vertex should be less than or equal to some finite number d. Accordingly, we consider only regular graphs, and we attempt to solve the problem of minimizing the diameter of regular graphs of degree d having, say, N vertices. There is an *a priori* upper bound to N, given d, called the Moore bound. (see Reference [2]), which is

$$N \leq 1 + d + d(d-1) + \cdots + d(d-1)^{k-1}$$

where k is the diameter of the graph. This says that asymptotically, k grows at least as fast as

$$\log_{d-1}(N) = \frac{\ln N}{\ln(d-1)}.$$

It is known, however, that this bound is only sharp in a finite number of cases for $d \geq 3$ (for d=2, the cyclic graphs are all examples where it is sharp), and it seems generally to be rather crude. Our efforts toward studying this problem consist of the construction of a heuristic algorithm (see Section 4.2.2), and some specific constructions derived from

group theory (see Section 4.2.4).

The diameter itself is only a weak measure of the effectiveness of the processor array. Suppose, for instance, that the processors have no memory, and that one wishes to transfer information simultaneously from the vertex v_i to the vertex w_i, for $i=1, \ldots, k$. Since there is no memory, one must produce a collection of paths $\phi^{(i)}$ of length n in Γ, with $v_M^{(i)} \neq v_M^{(j)}$ for all $1 \leq i,j \leq k$, and so that $v_0^{(i)}=v_i$, $v_n^{(i)}=w_i$. Here, $\phi^{(i)}=\{v_0^{(i)}, \ldots, v_n^{(i)}\}$. Such a collection of paths is called a k-separated multipath in Γ from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$. We define the k-separated distance between $(v_1, \ldots, v_k)$ and $(w_1, \ldots, w_k)$ to be the minimum length of all k-separated multipaths from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$ in Γ, and denote it by

$$d_k((v_1, \ldots, v_k),(w_1, \ldots, w_k)).$$

The k-separated diameter is

$$\max d_k((v_1, \cdots v_k),(w_1, \cdots, w_k)) = \Delta_k(\Gamma)$$

where the max is taken over all pairs of k-tuples, with $v_i \neq w_j$ unless $i=j$.

The k-separated diameter $\Delta_k(\Gamma)$ takes congestion into account, and so is a more sensitive measure of effectiveness of the processor array. However, it assumes that the processors have no memory. We will define another measure of efficiency $\Delta_k(\Gamma,l)$, which assumes that each processor has l units of memory. (Notice that this is not *equivalent* to having memory, but is simply a measure which attempts to take into account some possible improvements in algorithms which would be made possible by having memory.) By a (k,l)-separated multipath of length from $(v_1, \ldots, v_k)$ to $(w_1, \ldots, w_k)$, we mean a k-tuple $\phi^{(i)}$ of paths of length n, so that

a. $\phi^{(i)}$ is a path from v_i to w_i,

and

b. Each of the k-tuples $(v_j^{(1)}, v_j^{(2)}, \ldots, v_j^{(k)})$ contains each vertex at most l times.

Note that

$$\Delta_k(\Gamma,1) = \Delta_k(\Gamma),$$

and that for $l \geq k$, $\Delta_k(\Gamma,l) = \Delta(\Gamma)$. Intuitively, $\Delta_k(\Gamma,l)$ measures the time required to transfer the information in any k-tuple of processors to any other k-tuple of processors, given that each processor has l units of memory.

We now observe that for any graph Γ, $\Delta_k(\Gamma,l)$ is just the diameter of a graph associated with Γ. For, form the k-fold product graph

$$\Gamma \times \cdots \times \Gamma.$$

In $\Gamma \times \cdots \times \Gamma$, consider the full subgraph Γ_k' on the set $V_k' \subset V \times \cdots \times V$, (here V denotes the vertex set of Γ), where

$$V_k^l = \{(v_0, \ldots, v_k) | \text{no } v_j \text{ appears more than } l \text{ times}\}.$$

Then it is easy to see that

$$\Delta_k(\Gamma, l) = \Delta(\Gamma_k^l).$$

Thus, we have reduced these more sophisticated measures of effectiveness to a diameter question; this will be a useful reduction in view of the algorithm to be considered in the next section.

Unfortunately the diameter is often rather expensive to compute. Consequently, one would like to obtain some less sensitive, more easily computable invariants of graphs which still have some relation to the diameter.

Definitions: A *cycle* is a path from some given node to itself. The *girth* of a graph Γ is the length of the shortest cycle of Γ which contains no repeated edges.

We note that if a graph has diameter k, then its girth, is at most, $2k+1$. Intuitively, the girth is inversely related to the diameter. For a fixed number of points, small diameter tends to imply large girth. Let us summarize the facts known about girth relating to this problem.

a. There is a lower bound for the number of points in a graph with a given girth, analogous to the Moore bound (see Reference [3]).

b. For a graph of odd girth $g=2k+1$, and degree d, the number of points is at least

$$1 + d + d(d-1) + \cdots + d(d-1)^{k-1}.$$

This bound is obtained only for $d = 2$, or for $d = 3,7$, and possibly 57, with $g = 5$.

c. For a graph of even girth $g = 2k$ and degree d, the lower bound is

$$1 + d + D(d-1) + \cdots + d(d-1)^{k-2} + (d-1)^{k-1}.$$

This bound is known to be attainable only for $g = 4, 6, 8$, or 12. Also, only $d = p^s+1$ are known to occur, where p is a prime. In each case the diameter is k. These graphs provide by far the optimal graphs for the diameter problem with given diameter and degree.

We define certain numbers related to the girth. Given a vertex $x \in \Gamma$, we define $N_k(x)$ to be the number of vertices connected to x by a path of length k, and define $\nu_k(\Gamma)$ to be

$$\min_{x \in \Gamma} N_k(x).$$

Note that if Γ is a regular graph of degree d, then $\nu_k(\Gamma)$ is bounded above by $1+d+d(d-1)+\cdots+d(d-1)^{k-1}=\mu_k(d)$, and that the girth of Γ is $\geq l$ if $\nu_k(\Gamma)=\mu_k(d)$ for all $k \leq l/2$. Consequently, to maximize girth, one should attempt to maximize successively $\nu_2(\Gamma), \nu_3(\Gamma), \ldots, \nu_k(\Gamma), \ldots$. In each case, $\nu_{k+1}(\Gamma)$ should be maximized subject to the constraint

that one remains at an optimum for the previous values of the subscript. ν_k is quickly computed for small values of k, and the ν_ks are another more computable collection of invariants of graphs, which are related to the efficiency of the associated array of processors. This is particularly useful in attempting to work with the measures $\Delta_k(\Gamma,l)$; since the graphs Γ_k^l are usually quite large, the time spent computing invariants is of primary importance.

4.3.2 A Heuristic Algorithm.

A "hill-climbing" algorithm is produced here, using a particular heuristic criterion to find graphs of a given fixed degree and number of points with small diameter.

Let Γ be a graph of degree d, and let v_1,v_2,w_1,w_2 be vertices of Γ so that v_1v_2 and w_1w_2 are edges of Γ. Then by the *perturbed graph* based on (v_1,v_2,w_1,w_2), we mean the graph $\tilde{\Gamma}$ whose vertices are the same as those of Γ, and whose edge set is $(E_\Gamma-\{v_1v_2,\ w_1w_2\})\bigcup\{v_1w_1,\ v_2w_2\}$. Here E_Γ is the edge set of Γ. We say also that $\tilde{\Gamma}$ is the result of a *perturbation* on Γ. These modifications are precisely the X-changes defined in Reference [4]. $\tilde{\Gamma}$ is regular of degree d if Γ is. We will view these perturbations as "small" change in the graph, and move in directions which improve a certain functional which we define below. Graphs will be encoded by their "incidence matrices." We number the vertices of the graph Γ, by $\{v_1,\ldots,v_N\}$. By the incidence matrix $A(\Gamma)$ we mean the matrix (a_{ij}), where $a_{ij}=1$ if v_iv_j is an edge of Γ or $i=j$, and $a_{ij}=0$ otherwise. One useful property of $A(\Gamma)$ is:

The (i,j)-th entry of $A(\Gamma)^k$ is the number of paths of length $\leq k$ from v_i to v_j in Γ.

Consequently, the diameter of Γ is the least value of k for which all the entries of $A(\Gamma)^k$ are non-zero. This criterion is used in the algorithm to compute the diameter, since matrix powers are readily computable by a machine.

The diameter alone is itself not a sufficiently sensitive invariant for purposes of the algorithm. Specifically, there are too many graphs for which no perturbation results in an improvement of the diameter. Consequently, using only the diameter as a functional to be optimized, the algorithm is frequently unable to find a graph with even reasonable diameter. A definition is needed to improve matters. Let Z denote the integers. We wish to define an ordering on Z^n, called the *lexicographic* ordering. For $n=1$, the lexicographic ordering on $Z^n=Z$ is just the usual ordering on the integers. For $n>1$, we suppose the ordering is already defined for all $m<n$. We write $Z^n=Z\times Z^{n-1}$, and define the ordering inductively by

$$(z,w)<(z',w') \text{ iff} \begin{cases} z<z' \text{ or} \\ z=z' \text{ and } w<w' \end{cases}$$

for $z\in Z, w\in Z^{n-1}$. Now, we associate to every graph its "diameter vector." First, for a positive integer l, we define $\alpha_l(\Gamma)$ to be the number of zeros in the l^{th} power of $A(\Gamma)$. Of course, if $l>\Delta(\Gamma)$, $\alpha_l(\Gamma)=0$. The diameter vector is now simply the vector

$$(\Delta(\Gamma)\,(=k),\ \alpha_{k-1}(\Gamma),\ \alpha_{k-2}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\,).$$

We will denote this by $V(\Gamma)$. We order these vectors as follows:

$$(\Delta(\Gamma),\ \alpha_{k-1}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\,) \leq (\Delta(\Gamma'),\ \alpha_{k'-1}(\Gamma),\ \ldots,\ \alpha_2(\Gamma)\,)$$

if, and only if,

$$\left\{\begin{array}{c}\Delta(\Gamma) < \Delta(\Gamma') \ \ or \\ \Delta(\Gamma) = \Delta(\Gamma') \ \text{and} \ \ \alpha(\Gamma) \leq \alpha(\Gamma')\end{array}\right\}.$$

Here, $\alpha(\Gamma)$ denotes the vector $(\alpha_{k-1}(\Gamma),\ \ldots,\alpha_2(\Gamma))$, and the ordering is the lexicographic one.

The algorithm now proceeds as follows. A 4-tuple (v_1,v_2,w_1,w_2) is Γ ***admissible*** if v_1v_2 and w_1w_2 are edges of Γ, and v_1w_1 and v_2w_2 are not. To a Γ-admissible 4-tuple, we may associate a perturbation of Γ, as defined above. From a fixed initial graph Γ, Γ-admissible 4-tuples are generated, and the associated perturbations are applied. This continues for a large number of steps, until the initial graph is presumed randomized. The 4-tuples are generated using a random number generator. The fixed initial graph (in the trivalent case) is an n-cycle with antipodal points connected. After this is done, the steps are as follows:

a. Select an Γ-admissible 4-tuple at random

b. Compute $V(\tilde{\Gamma})$, where $\tilde{\Gamma}$ is the graph obtained by applying the perturbation associated to the 4-tuple constructed in (a). If $V(\tilde{\Gamma}) < V(\Gamma)$, set $\Gamma = \tilde{\Gamma}$. Repeat step (a).

The perturbations are selected at random, since it was found that a simple ordering of perturbations tended to bias the algorithm toward particular graphs.

We compare our algorithm to that devised in Reference [4]. Our perturbations are precisely their X-changes, but the functional we optimize is much more sensitive. Theirs consists only of $\Delta(\Gamma)$ and of $\alpha_{k-1}(\Gamma)$.

Summarizing the results of the application of our algorithm, by the use of the algorithm, it has been possible to improve substantially most of the densest known graphs. We give our improved version of the table constructed in Reference [5]. d denotes the degree of the graph, k the diameter. The (d,k) entry is the largest known graph with diameter k and degree d. Our entry is listed above; the parenthesized value below is the value from Reference [5]. One asterisk indicates that the graph is provably optimal. Two asterisks indicates that it is obtained from Reference [3] using the result cited in Section 4.1.

The results obtained from this algorithm are in some cases surprising. Some of the qualitative properties we observed are:

a. Many graphs obtained by random generation of graphs improved values in the older version of the table in Reference [5] given by Storwick [6]. This suggests that one is further from optima than was previously thought.

b. By evaluating the eigenvalues of the incidence matrices arrived at by the algorithm, it was found that there are many distinct "local minima" (i.e., graphs for which no perturbation improves the diameter vector) for the diameter vector. This contradicts the suggestion made in Reference [4], that one tends to arrive at a global optimum from all starting points. It seems that the algorithm in Reference [4] suffers from two deficiencies. First, their objective functional for minimization is not sufficiently sensitive, as we observed above. Second, their perturbations are done in fixed sequential order, which severely skews their results. We have overcome this difficulty by randomly selecting the perturbations at each stage.

c. Although our algorithm is efficient, it seems that substantially larger networks could be studied if our diameter routine were modified to use the so-called "Dijkstra algorithm," which would speed up the diameter calculation substantially.

d. Although the algorithm is an improvement over all previous heuristic algorithms for this problem, it is unable to find many known dense graphs, arising from systematic constructions. The reason for this seems to be the "denseness" of the set of local optima in the set of all graphs of degree d, and the relative sparseness of the so-called vertex transitive graphs therein. It seems, therefore, that it would be desirable to design an algorithm which operates entirely inside a collection of vertex transitive graphs, possibly with the Cayley graphs (see Section 4.2.4). Using the Dijkstra diameter algorithm and an efficient description of many groups, such an algorithm should be constructible. Moreover, it would allow much larger networks to be studied, since the diameter calculation for vertex transitive graphs is substantially shorter than that for arbitrary graphs.

d\k	2	3	4	5	6	7	8	9	10
3	10* [10]	20* [20]	38 [34]	58 [56]	126** [84]	160 [122]	240 [176]	400 [311]	600 [525]
4	15* [15]	36 [35]	80 [67]	150 [134]	728** [261]	728** [425]	910 [910]	2520 [1360]	2888 [2312]
5	24* [24]	56 [48]	170** [126]	300 [262]	2730** [505]	2730** [1260]	2730** [2450]	5760 [4690]	9648 [9380]
6	31 [31]	80 [65]	312** [164]	600 [600]	7812** [1152]	7812** [2520]	7812** [6561]	19683 [19683]	59049 [59049]
7	50* [50]	100 [88]	312** [252]	992 [992]	7812** [2850]	7812** [4680]	12960 [12250]	43200 [43200]	90000 [86400]
8	57 [57]	140 [105]	800** [384]	2550 [2550]	39216** [5760]	39216** [16384]	65536 [65536]	262144 [262144]	1048576 [1048576]
9	74 [74]	160 [150]	1170** [600]	3306 [3306]	74898** [12500]	74898** [20160]	76500 [76500]	382500 [382500]	1048576 [1048576]
10	91 [91]	250 [200]	1640** [864]	5550 [5550]	132860** [25000]	132860** [78125]	390625 [390625]	1953125 [1953125]	9765625 [9765625]

(degree = d, diameter = k)

Figure 4.1. Densest known regular graphs, June 1983.

4.3.3 Modification of the Algorithm for More Sensitive Measures.

In view of the remarks in Section 4.1 which identify the measures $\Delta_k'(\Gamma)$ as the diameter of an associated graph Γ_k', one can study these measures in principle using the algorithm discussed in Section 4.3.2. However, the graphs Γ_k' are usually too large for this procedure to be practicable. Our current implementation of the algorithm will accept only graphs with fewer than 1000 points, and Γ_k' usually is larger than this. For the measures Δ_k', we, therefore, propose the use of a "dual algorithm" based on girth, which is much simpler to compute. (See Section 4.3.1.)

The modified girth algorithm is identical to the previous algorithm except that the objective functional is altered. For a graph Γ, we define its girth vector to be

$$\gamma(\Gamma)=(\nu_1(\Gamma),\nu_2(\Gamma), \ldots ,\nu_k(\Gamma),\ldots).$$

This is ordered by the lexicographic ordering, and the algorithm proceeds just as before, except that we now accept a perturbation if it increases $\gamma(\Gamma)$. Applying this algorithm to Γ_k' should produce heuristic results which improve these measures.

4.3.4 Vertex-Transitive Graphs.

Two desirable features of a graph to be used as a processor array are that the description be as simple and as compact as possible. Thus, the graphs produced by a heuristic algorithm generally will not be satisfactory from this point of view. For this purpose it would be useful to restrict oneself to a class of graphs having a compact description.

One such family is the collection of so-called *Cayley graphs*. (See Chapter 2 for definitions and notation.) As an example, if $G=Z_n$, the cyclic group with n elements, and $\Omega=\{T,T^{-1}\}$, where T is a generator of G, the associated graph is the cyclic graph of size n. A useful property of $\Gamma(G,\Omega)$ is that its automorphism group is transitive on the vertices. This is clearly the case since the right G-action on G provides an action of G on the graph, which is clearly transitive on the vertex set. This is a useful property, since it means that the diameter may be computed by finding the points of maximal distance from one given point. Also, routing algorithms for these networks are compactly described, since one must only find optimal paths starting at one given point.

One important proposed architecture, the cube-connected cycles of Reference [1], is of this form. In fact, if G is the semi-direct product $Z/n\underset{\rho}{\times}(Z/2)^n$, where if T is a generator for Z/n, $\rho(T)(x_1,\ldots,x_n) = (x_n,x_1,\ldots,x_{n-1})$. Thus, G has elements (m,v), where $m\epsilon Z/n$, $v\epsilon(Z/2)^n$, and $(m,v)(m',v') = (m+m',\rho(m')v+v')$. It is an easy calculation to see that if $\Omega = \{(1,0), (-1,0), (0,e_1)\}$, where $e_1 = (1, 0, 0 ,\ldots, 0)$, then $\Gamma(G,\Omega)$ is, in fact, isomorphic to the graph associated with the cube-connected cycles. The diameter of the cube-connected cycles is known to grow as $\frac{5}{2}\log_2(K)$, where K is the number of vertices in the graph.

We should remark here that large girth (and hence small diameter) in Cayley graphs is associated with non-commutativity of the group in question. This being the case, the simple groups seem to be natural candidates to produce efficient graphs. This is proven in studying the diameters of $\Gamma(G,\Omega)$ for certain choices of Ω, and $G=S_n$, and observing that for large n, they approximate the Moore bound. The order of symmetric group S_n is $n!$. Let $\Omega_k\subset S_n$, be the set of all cycles of length $\leq k$. We propose to compare the diameter of $\Gamma(S_n,\Omega_k)$ with its associated Moore bound. First, we observe that

$$|\Omega_k| = \binom{n}{2} + 2\binom{n}{3} + \cdots + (k-1)!\binom{n}{k}$$

by a simple counting argument. Thus, the degree of $\Gamma(S_n,\Omega_k)$ is

$$d = \sum_{j=1}^{k} (j-1)!\binom{n}{j}.$$

Thus, for large n, the Moore bound for $\Gamma(S_n,\Omega_k)$ is

$$\log_{d-1}(n!) = \frac{\ln(n!)}{\ln(d-1)}.$$

By Stirling's formula,

$$\lim_{n\to\infty} \frac{\ln (n!)}{n \ln n} = 1,$$

so for large n, the Moore bound is approximated by

$$\frac{n \ln n}{\ln (d-1)}.$$

But, again for large n,

$$\ln (d-1) = \ln \left(-n + \sum_{j=1}^{k} (j-1)! \begin{bmatrix} n \\ j \end{bmatrix} \right)$$

is approximated by $\ln \begin{bmatrix} n \\ k \end{bmatrix} \approx k \ln n$. Hence, for large n, the Moore bound for $\Gamma(S_n, \Omega_k)$ is approximated by n/k. The diameter of $\Gamma(S_n, \Omega_k)$, on the other hand, tends to $n/k - 1$, as one readily computes in S_n. Consequently, the diameter of $\Gamma(S_n, \Omega_k)$ is within a factor of

$$\frac{k}{k-1} = 1 + \frac{1}{k-1}$$

of the Moore bound. As k becomes larger, we are able to approximate close equality arbitrarily. So it seems that S_n is a plausible candidate for further study.

We now show how to modify the cube-connected cycles, using group theoretic methods, to provide an infinite family of vertex transitive trivalent graphs, whose diameter is substantially smaller, but which has all the desirable regularity properties of the cube-connected cycles.

Let $G_n = Z/n \underset{\rho}{\times} (Z/2)^n$, as before. Note that G_n contains a central element, namely the vector $(0,(1, 1, \ldots, 1))$. Following our intuition concerning the relationship between noncommutativity and small diameter, we eliminate the central element by simply factoring it out. Call the quotient group $\tilde{G}_n$, and let $\tilde{\Omega}$ denote the image of Ω. Then we claim that the diameter of $\Gamma(\tilde{G}_n, \tilde{\Omega})$ grows as $2\log_2(k)$, where k is the number of vertices, an improvement over the diameter $\frac{5}{2}\log_2(k)$, obtained for the cube-connected cycles.

Proposition: The diameter of $\Gamma(\tilde{G}_n, \tilde{\Omega})$ grows as $2\log_2(k)$.

Proof. By taking inverses, we can clearly consider the graph

$$\vec{\Gamma}(\tilde{G}_n, \tilde{\Omega}),$$

where $(g, g\omega)$ is an edge for $\omega \in \Omega$. An element of $\tilde{G}_n$ is given by an ordered pair (m, v), where

$$m \in Z/n, \quad v \in \bar{V}(Z/2)^n/(1,\ldots,1).$$

The multiplication is given by

$$(m,v)e_1 = (m,v+e_1), \ (m,v)T = (m+1,\rho(T)v), \ \text{and} (m,v)T^{-1} = (m-1,\rho(T)^{-1}v).$$

An algorithm for expressing (m,v) in terms of the generators T, T^{-1}, and e_1 is described as:

Let $x_1(\tilde{v})$ denote the first coordinate of a vector $\tilde{v} \in V=(Z/2)^n$. For any $v \in \bar{V}$, one may lift v to an element $\tilde{v}$ of V, so that the number of non-zero coordinates in $\tilde{v}$ is $\leq n/2$, for if one lifted $\tilde{v}$ does not have this property, then $\tilde{v}$ + (1, 1, ..., 1) does. Given (m,v), select $\tilde{v}$ as above.

The algorithm now proceeds as follows:

a. Initialize a counter α at $n-1$.

b. Is $x_1(\tilde{v}) = 0$? If yes, proceed to (d), if no, proceed to (c).

c. Multiply by e_1. Proceed to (d).

d. Multiply by T, and decrement α by 1. Proceed to (e).

e. Is $\alpha = 0$? If so, proceed to VI. If not, return to (b).

f. If $m \neq 0$, multiply by T^q, where q is the number of minimal absolute value congruent to $-m$ mod n. Note that $|q| \leq \frac{n}{2}$. Quit.

Since $\tilde{v}$ has at most $n/2$ 1s, we only multiply by e_1 at most $n/2$ times. Thus, the total number of steps is at most $(n-1)+n/2+n/2 = 2n-1$. For odd n, it is at most $2n-3$, which is of the same order as $2\log_2(n2^{n-1})$.

If one forms the quotient of this graph by the equivalence relation $(m,v) \approx (m',v)$ for all $m,m' \in Z/n$, one obtains a non-regular family of graphs whose diameter grows as $\frac{3}{2}\log_2(n)$, which is comparable to that obtained in Reference [7], and for which the routing algorithm is much simpler. Finally, an alternative version of this construction is given by forming the

$(n-1)$-cube, inserting n-cycles at every vertex so that the incoming edges each connect at distinct vertices, and connect the remaining vertex to the corresponding vertex for the antipodal point on the cube.

4.3.5 A Layout for the Modified Cube-Connected Cycles.

In the paper Reference [1], two layouts are proposed for the cube-connected cycles, one slightly more efficient than the other. By combining these two layouts, we obtain a layout for the modified cube-connected cycles. The area of the layout grows as 3/2 times the area of the cube-connected cycles with the same number of nodes, and has communication time roughly 4/5 times that of the cube-connected cycles. We give the layout for the case n=5, corresponding to $5 \cdot 2^4 = 80$ nodes. It is clear from the diagram (Figure 4.2) how to extend to the general case.

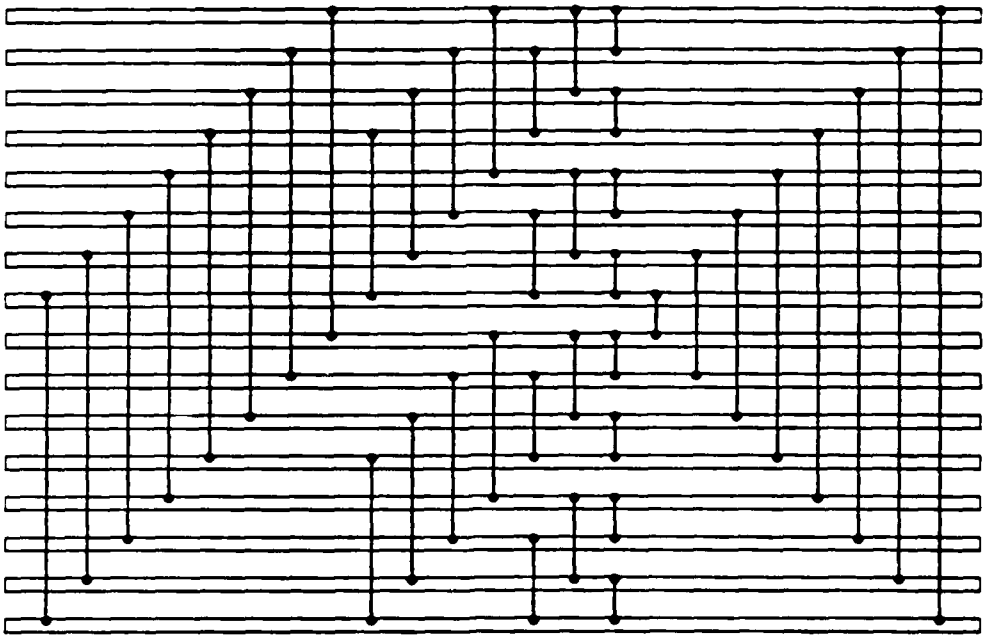


Figure 4.2. A layout for the modified cube-connected cycles

4.4 References

[1]. Preparata, F. and Vuillemin, J., "The cube-connected cycles: a versatile network for parallel computation," *Communication of the ACM*, Vol 24, No. 5, May 1981.

[2]. Bolloba´s, B., *Extremal Graph Theory*, Academic Press, 1978.

[3]. Biggs, N., *Algebraic Graph Theory*, Cambridge University Press, 1974.

[4]. Toueg, S. and Steiglitz, K., "The design of small-diameter networks by local search," *IEEE Transactions on Computers* 28(7), 1979.

[5]. Leland, W., Finkel, R., Qiao, L., Solomon, M., and Uhr, L., "High density graphs for processor interconnection," Information Processing Letters, Vol. 12, No. 3, 1981.

[6]. Storwick, R. M., "Improved construction techniques for (d,k) graphs," *IEEE Transactions on Computers*, 19(12), 1970.

[7]. Leland, W. and Solomon, M., "Dense trivalent graphs for processor interconnection," *IEEE Transactions on Computers*, Vol C-31, No. 3, Mar 1982.

5. Modular Hardware Description Language

5.1 Introduction

This chapter describes a modular hardware description language (MHDL) developed to provide an easy means of simulating the numerical, and other high-level, behavior of novel computer architectures, especially those proposed for real-time signal processing applications. The principle goals of MHDL are:

a. Easy specification of elementary building blocks (machines) at an algorithmic level.

b. Automatic reproduction of any number of already designed modules and easy specification of interconnection schemes for these modules to produce new modules.

c. The behavior and performance of the resulting machines simulated by the compiled MHDL code.

This language was developed on and for computer systems using UNIX[11] operating systems. While it could be modified to run on any system supporting the C programming language, only its use on UNIX systems is discussed here, and make use of programs available on UNIX with little or no comment.

The remaining sections of this chapter give a brief description of the procedure for installing MHDL on a system (which may be in practice less than automatic due to differences in C compilers even among "standard" UNIX systems), and for compiling MDHL programs. We also describe the syntax and grammar of the language, and discuss possible future improvements of the language. Examples of MHDL programs and source listings for the compiler may be found in Reference [1].

5.2 MHDL

5.2.1. Components of the Compiler

The compiler consists of the following files:

lexer.c

This is the source file for the lexical analyzer for MHDL. It breaks the input stream from a MHDL program into tokens, and stores all identifiers in a symbol table. Although *lex* was not used for this lexical analyzer, much of the structure of the lexical

[11] UNIX is a registered trademark of Bell Laboratories.

analyzer remains compatible with the lex environment.

mhdlyacc

This is the source file for the MHDL parser and code generator. It receives the input stream and tokens from the lexical analyzer, checks for any syntax errors, and generates appropriate code in the language C. Since the syntax for MHDL is very straightforward, only a modest number of error messages have been included in the parser.

ytab.c
y.tab.h

These are files produced when yacc is run on mhdlyacc.

declar.h

This contains all the global declarations for the combined program of lexer.c with mhdlyacc.

Makefile

This is the makefile for mhdl. The command *make mhdl* will create the file *mhdl* which contains the object file for the compiler. Note that the files *Makefile*, *declar.h*, *mhydlyacc*, and *lexer.c* must all be present to be able to "*make*" mhdl.

mhdl

This is the object file for the MHDL compiler. It is produced by Makefile using the command *make mhdl.*

xxmhdl.c
xxmhdl.global
xxmhdl.proced
xxmhdl.declar
yymhdl.c

These files are all produced when mhdl compiles a MHDL program. The four files of the form xxmhdl.* are always produced by mhdl. The file yymhdl.c, which is simply a readable version of the xxmhdl files, is only produced if the mhdl compiler detects no errors. Note that if the C compiler finds errors in the program it is much easier, if not completely necessary, to work with yymhdl.c rather than the xxmhdl.* version.

5.2.2. Using the Compiler

A file called *mhdl* is needed to compile a MHDL program. If this file does not exist on your system, then see the discussion in Section 2a to obtain a copy of this file. The steps for using MHDL are the following:

a. Enter *mhdl MHDLprogram* (a MHDL program may consist of several files).

b. Correct all errors reported by the MHDL compiler and repeat step a.

c. Enter *cc yymhdl.c*.

d. Correct all errors reported by the C compiler using the file *yymhdl.c* for reference, and repeat steps a, b, and c.

e. Enter *a.out* or *a.out < datafile* depending on whether you wish to use standard input or an already prepared input file. (The string "*datafile*" shouldn't be interpreted as a literal.)

5.3 Description of the Language

5.3.1 Syntax and Grammar

A MHDL program is made up of a sequence of blocks. Each block may be any one of the following types:

Primitive Module
Module
Global
Procedure
Connection Scheme
Configuration

The syntax for each of these blocks is illustrated below:

Primitive Module <*module name*>

Parameter <*var. decls.*>
Input <*var. decls.*>
Output <*var. decls.*>
Inout <*var. decls.*>
State <*var. decls.*>
Uses
  Procedure <*procedure names*>
End Uses
Behavior
  <*C code*>

End Primitive Module <*module name*>

Module <*module name*>

Parameter <*var. decls.*>
Input <*var. decls.*>
Output <*var. decls.*>
Inout <*var. decls.*>
Uses
Primitive Module <*prim. mod. name* [*no. of times used*]>
Module <*mod. name* [*no. of times used*]>
Connection Scheme <*connection scheme name*>
Configuration <*configuration name*>
Procedure <*procedure name*>
End Uses
Behavior
<*C code*>

End Module <*module name*>

Global
<*C code*>
End Global

Procedure <*procedure name*>
<*C procedure*>
End Procedure <*procedure name*>

Connection Scheme <*scheme name*>
End Connection Scheme <*scheme name*>

Configuration <*configuration name*>
End Configuration <*configuration name*>

The following is slightly informal Backus-Naur form for the grammar for MHDL. Literals are in **boldface**. Alternatives are separated by a vertical bar "|". A group that may be repeated a certain number of times is enclosed in braces, "{" and "}", with the number of repetitions indicated by "+" to indicate 1 or more repetitions and a "*" to indicate 0 or more repetitions. Optional terms are enclosed in "[" and "]", and any

terms starting with *C-* are meant to refer to the corresponding objects in the language C.

program → { block } +

block →

{ prim-module-block | module-block | global-block | procedure-block | connect-scheme-block | config-block | whitespace }

prim-module-block →

Primitive whitespace **Module** whitespace block-name whitespace
var-declarations
State *C-code*
[**Uses** { **Procedure** identifier }* **End** whitespace **Uses**]
Behavior *C-code*
End whitespace **Primitive** whitespace **Module**
whitespace block-name

module-block →

Module whitespace block-name whitespace
var-declarations
Uses
{ { { **Primitive** whitespace **Module** | **Module** }
{ whitespace identifier [*C-code*] }+ } |
Procedure whitespace {identifier}+ |
Configuration whitespace {identifier}+ |
Connection whitespace **Scheme** whitespace {identifier} + } +
End whitespace **Uses**
Behavior *C-code*
End whitespace **Module** whitespace block-name

global-block → **Global** *C-code* **End** whitespace **Global**

procedure-block →

Procedure whitespace block-name *C-code*
End whitespace **Procedure** whitespace block-name

connect-scheme-block →

Connection whitespace **Scheme**
whitespace block-name *C-code*
End whitespace **Connection** whitespace block-name *C-code*

config-block →

Configuration whitespace block-name *C-code*
End whitespace **Configuration** whitespace block-name
block-name → *C-identifier*

var-declarations →

[**Parameter** *C-variable-declarations*]
[**Input** *C-variable-declarations*]
[**Output** *C-variable-declarations*]
[**Inout** *C-variable-declarations*]

whitespace → { *C-whitespace* | MHDL-comment }+
MHDL-comment → $ {any character except *NEWLINE* or *FORM-FEED*}*

5.3.2 MHDL Semantics - How MHDL 'runs' a Module

The Module and Primitive Module blocks are the only ones in this version of MHDL that have nontrivial behavior. Configuration and Connection Scheme blocks are unsupported in this version and cause an error message. The C-code in Global and Procedure blocks is copied directly to sections of the produced code external to all other procedures. The code in a Global block is guaranteed to appear before all other codes.

The two kinds of modules, primitive and non-primitive, are set up in very different ways. For primitive modules all of the input and output variables, together with the state variables, are put together as one structure declaration. The number of times this primitive module is used in the machine being described is counted and an array of glo-

bal variables is declared with the type of the structure just created. As the global machine runs, bookkeeping is done to keep track of the index of the current primitive module that is actually running. Only the I/O and state variables for that particular copy of the primitive module are affected. Running a primitive module amounts to executing the code in the Behavior section exactly as it appears, except that all I/O and state variables are preceded with an array structure pointer.

For non-primitive modules all of the input and output variables are used to create a global variable in the same manner as what was done for primitive modules. Bookkeeping for the current running copy of the module is also done in the same way as for primitive modules. Running a non-primitive module should be thought of as occurring in two steps. In the first step, the Behavior section of the module is executed as it appears, and has this module to various inputs of its submodules. For the second step, the Uses declaration is used to count the number of times a submodule is used to make up this module, and the submodule is simply run that many times, incrementing the index of the submodule for each run. There is a distinguished module with name "Main" that is always the module representing the global machine. The program created by MHDL has as its only task the running of Main until something in the MHDL program causes the program to terminate (usually caused by executing a *exit()* in one of the primitive modules).

5.4 Possible Improvements

The present version of MHDL was designed as a prototype and as such many possible extensions or changes were not incorporated until more experience had been gained with language. The following list of changes contains features that the designer would most like to see improved. Many of these features may not appeal to general users, and many may no longer be appropriate if the intended use of this language should shift.

a. Allow levels of nesting of modules.

The current version allows only one level of nesting, and this is clearly too restrictive for general use on larger problems.

b. Allow Global Declarations within Modules

The language should promote module structure by not forcing the user to put global declarations in a separate block.

c. Increase debugging facilities

(1) Test Connection Structure for 0 or Multiple Connections.
(2) Check that a module's I/O variables are only used in an appropriate way (e.g., that Input variables are only used on the right sides of equations).
(3) Allow easy (or automatic) printout of the values for I/O and state variables

to aid user debugging.

d. Allow Identifiers to use any Number of Significant Letters.

e. Develop Connection Scheme Concept Beyond linear Numbering.

f. Allow Multiple and Conditional Calls of Modules.

The multiple calls would be useful for example when a module operating at the word level uses a module operating at the bit level. Conditional calls could be useful if certain electronic characteristics wished to be simulated at the MHDL level (rather than hidden in the user's code).

g. Develop Parameter Concept for Modules.

h. Improve Initialization Facilities

Several types of initialization are now awkward within MHDL and could be greatly improved. Currently all I/O and state variables are initialized to the value 0, and there is no mechanism for other initialization values. Along the same lines, files need to be opened for use and currently only standard input and output are easily accessed.

5.5 References

[1]. Wright, C. G., "Modular Hardware Description Language," Computer Sciences Corporation TR, 20 September 1983.